



Europäisches Patentamt
European Patent Office
Office européen des brevets



Publication number: 0 529 913 A2

(12)

EUROPEAN PATENT APPLICATION

(21) Application number: 92307532.9

(51) Int. Cl.⁵: G06F 9/38

(22) Date of filing: 18.08.92

(30) Priority: 26.08.91 US 750132

(43) Date of publication of application:
03.03.93 Bulletin 93/09

(84) Designated Contracting States:
DE FR GB

(71) Applicant: INTERNATIONAL BUSINESS
MACHINES CORPORATION
Old Orchard Road
Armonk, N.Y. 10504 (US)

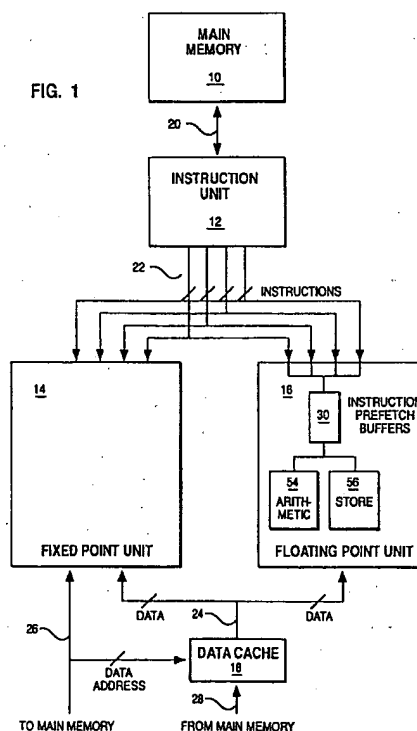
(72) Inventor: Fry, Richard Edmund
200 East Nakoma
Round Rock, Texas 78664 (US)
Inventor: Hicks, Troy Neal
12804 Timberside Drive
Austin, Texas 78727 (US)

(74) Representative: Killgren, Neil Arthur
IBM United Kingdom Limited Intellectual
Property Department Hursley Park
Winchester Hampshire SO21 2JN (GB)

(54) Data processing system with multiple execution units.

(57) A data processing system is provided that includes a plurality of execution units 14,16 each including independent circuits for storing (30) and executing (54) instructions. A circuit (22) is also included for providing instructions from a sequence of instructions to the execution units where each instruction is provided to only one of the execution units. The system includes a circuit for detecting when an instruction in a first execution unit must complete execution prior to execution of an instruction in a second execution unit to produce correct results. A circuit is further included, responsive to the circuit for detecting, for delaying executing the instruction in the second execution unit until the instruction in the first execution unit has completed execution.

FIG. 1



EP 0 529 913 A2

This invention relates to data processing systems and, more specifically, to the parallel execution of instructions out of sequence in a data processing system with multiple execution units.

Data processing systems have historically required instructions to be executed sequentially. It is, of course, advantageous to execute instructions in a data processing system as quickly as possible. One approach in the prior art has been to speed up the performance of an existing unit by increasing its throughput. A second approach has been to use multiple execution units and execute instructions in parallel as much as possible. When executing instructions in parallel in multiple execution units, it is necessary to provide a method to handle data dependencies between instructions.

A method for handling data dependencies in a non-parallel data processing system is disclosed in U.S. Patent No. 4,630,195, entitled "Data Processing System with CPU Register to Register Data Transfers Overlapped with Data Transfer to and from Main Storage". This method uses tags on registers to determine whether a register to be used by an instruction is the subject of a pending I/O instruction. If the register is free, the instruction can execute without waiting for the I/O instruction to complete. This method, however, does not involve a parallel processing scheme.

One method to handle data dependencies in a parallel data processing system is to express programs according to data flow. An example of this method is disclosed in an article in IEEE Transactions on Computers, Vol. C-26, No. 2, February 1977, pages 138-146, entitled "A Data Flow Multi-processor". Instructions are separated into modules according to the operands each requires. If an instruction is dependent upon a second instruction, it is placed in the same module as the second instruction. Each module is self-contained and produces no side effects because all data dependencies are included. Thus, multiple processors are free to concurrently execute modules. The drawback of the method is that it requires a unique data flow language that is quite different from conventional systems.

A second method for handling instructions in a parallel data processing system is to synchronize the processors. An article in the IBM Technical Disclosure Bulletin, Vol. 32, No. 7, December 1989, pp. 109-113, entitled "Device for Synchronizing Multiple Processors" discloses a device for synchronizing multiple processes. The device is capable of barrier synchronization and of serialization of multiple requests from processes. Barrier synchronization is a point in a sequence of instructions that all processes must reach before any process can pass the barrier. A serialization operation is an operation that assigns a unique integer to each of multiple simultaneous requests to indicate priority or to assign each process a unique set

of system resources. This method suffers from the fact that it is too rigid and does not allow instructions to execute out of their original sequence.

U.S. Patent No. 4,763,294, entitled "Method and Apparatus for Floating Point Operations", discloses an apparatus for synchronizing a fixed point processor and a floating point processor depending upon the type of instruction. The floating point instructions are either a member of a first group of instructions requiring interlock or a second group not requiring interlock. In either case, the fixed point unit controls the dispatch of floating point instructions and must wait for the floating point processor to be idle. Thus, the fixed point processor always sees the instructions in their original sequence.

U.S. Patent No. 4,916,606, entitled "Pipelined Parallel Data Processing Apparatus for Directly Transferring Operand Data between Preceding and Succeeding Instructions", discloses an apparatus that detects when an instruction will use the result of a previous instruction and provides the data directly to the succeeding instruction. This speeds execution by vitiating the step of retrieving data but does not allow parallel processing instructions out of sequence.

U.S. Patent No. 4,956,800, entitled "Arithmetic Operation Processing Apparatus of the Parallel Processing Type and Compiler which is Used in this Apparatus", discloses an apparatus for performing arithmetic operation processes at a high speed by enabling the execution sequence and the input/output sequence in parallel. This speeds execution, but the instructions are still executed in sequence.

None of the references detailed above describe a mechanism that provides parallel execution of instructions out of sequence in independent execution units where an execution unit is delayed only when the correct execution of an instruction in one execution unit is dependent upon the completed execution of an instruction in a second execution unit.

According to the invention there is provided a data processing system, comprising: a plurality of execution units, each including independent means for storing and for executing instructions; means for providing instructions from a sequence of instructions to said execution units, where each of said instructions is provided to only one of said execution units; means, connected to said execution units, for detecting when an instruction in a first execution unit must complete execution prior to execution of an instruction in a second execution unit to produce correct results; and means, responsive to said means for detecting, for delaying executing said instruction in said second execution unit until said instruction in said first execution unit has completed execution.

In a preferred embodiment, a data processing system is provided in which the instructions belong to one of a plurality of classes of instructions and each execution unit is dedicated to executing one class of

instructions. The system further includes a circuit for determining the class of each instruction and a circuit for providing instructions belonging to a class to the execution unit dedicated to executing that class of instructions. A circuit is also included for associating with each instruction in a first execution unit the number of instructions in a second execution unit that preceded it in the original sequence. Finally, a circuit is included (1) for delaying executing an instruction in the first execution unit only if the count associated with the instruction is nonzero and an instruction in the second execution unit that preceded the instruction must execute prior to it to produce correct results, and (2) for delaying executing an instruction in the second execution unit only if the count associated with an instruction in the first execution unit is less than or equal to a number of instructions in said second execution unit that precede said instruction in said second execution unit and the instruction in the first execution unit must execute prior to the instruction in the second execution unit to produce correct results.

In order that the invention may be well understood, a preferred embodiment thereof will now be described with reference to the accompanying drawings, in which:

Fig. 1 is a block diagram illustrating the components of the data processing system;

Fig. 2 is a block diagram illustrating the floating point unit and its separate arithmetic and store execution units;

Fig. 3 is a flow chart illustrating the control of the instruction in the bottom position of the queue in the store instruction execution unit;

Fig. 4 is a flow chart illustrating the control of all the instructions in the queue in the store instruction execution unit;

Fig. 5 is a flow chart illustrating the control of the instruction in the bottom position of the queue in the arithmetic instruction execution unit;

Fig. 6 is a flow chart illustrating the control of all the instructions in the queue in the arithmetic instruction execution unit;

Fig. 7 is a block diagram illustrating an example of delaying an instruction in the bottom position of the queue in the store instruction execution unit;

Fig. 8 is a block diagram illustrating an example of executing an instruction in the bottom position of the queue in the store instruction execution unit out of sequence;

Fig. 9 is a block diagram illustrating an example of delaying an instruction in the bottom position of the queue in the arithmetic instruction execution unit and of executing the instruction in the second position of the queue in the arithmetic execution unit out of sequence;

Fig. 10 is a block diagram illustrating an example

of executing an instruction in the bottom position of the queue in the arithmetic instruction execution unit out of sequence; and

Fig. 11 is a block diagram illustrating an example of delaying an instruction in the bottom position of the queue in the store instruction execution unit and executing the instruction in the second position of the queue in the store execution unit out of sequence.

Description of the Preferred Embodiment

In the present invention, a data processing system is provided that is capable of independently executing instructions in separate execution units in the floating point unit. Consequently, instructions can be executed out of sequence. The separate execution units wait for each other only in the case where two instructions, in different execution units, must be executed in sequence because the correct result of one is dependent upon the completed execution of the other.

Fig. 1 is a block diagram of the data processing system of the preferred embodiment. The system includes a main memory (10), an instruction unit (12), a fixed point unit (14), a floating point unit (16), and a data cache (18). The main memory is connected to the instruction unit by the memory bus (20). The instruction unit is connected to the fixed point unit and floating point unit by the instruction bus (22). The floating point and fixed point units are interconnected by the data bus (24) and are connected to the data cache also by the data bus (24). The fixed point unit is connected to the main memory and to the data cache by the data address bus (26). The data cache unit is connected to main memory by the bus (28).

In the preferred embodiment, the instruction unit functions as local high speed storage for instructions. It takes instructions from the main memory and dispatches them to the fixed point and floating point units. The fixed point and floating point units synchronously execute their respective instructions. Within the floating point unit are instruction prefetch buffers (30), an arithmetic execution unit (54), and a store execution unit (56). The function of each of these elements is discussed with respect to Fig. 2. If data is to be stored to or loaded from memory, the fixed point unit places the address on the address bus (26). The data is transferred on the data bus (24). The data cache functions as local high speed storage for data and interfaces with main memory on the bus (28).

Fig. 2 is a block diagram of the floating point unit (16) of Fig. 1. The Instruction Prefetch Buffers (30) sequentially store instructions from the instruction bus (22) sent by the Instruction Unit (12 in Fig. 1). The Instruction Prefetch Buffers (IPB's) store instructions four at a time with the first instruction in sequence stored in IPB1 and the last in IPB4. The Arithmetic

Queue Control (36) and the Store Queue Control (38) select the instructions out of the IPB's and provide them to the Arithmetic Execution Unit (54) and the Store Execution Unit (56), respectively, on buses (50) and (52).

When an arithmetic instruction is moved into the Arithmetic Execution Unit, the Store Count Control (40) makes a count of the number of store instructions that precede it in the Store Execution Unit. The Store Count Control then sets the associated Store Count Field (24). The count made by the Store Count Control is called a "store count". The Store Count Field has one element to hold the store count associated with each arithmetic instruction in the Arithmetic Execution Unit. When a store instruction is executed, the Store Count Control decreases the store counts in the Store Count Field by one.

The Arithmetic Execution Unit (ArEU) includes the Arithmetic Instruction Queue (32), the Arithmetic Execution Logic (46), and the Arithmetic Execute Decision Control (42). Similarly, the Store Execution Unit (StEU) includes the Store Instruction Queue (34), the Store Execution Logic (48), and the Store Execute Decision Control (44). The Arithmetic Instruction Queue (AQ) and the Store Instruction Queue (SQ) store the instructions that are provided to the ArEU and the StEU, respectively. The Arithmetic Execute Logic (ArEL) and the Store Execute Logic (StEL) execute the instructions. The Arithmetic Execute Decision Control (ArEDC) and the Store Execute Decision Control (StEDC) function to control when an instruction in the AQ and the SQ can be executed. The ArEDC provides an execute signal on line (54), and the StEDC provides an execute signal on line (55). Instructions are transferred to the ArEL and the StEL on buses (53) and (55), respectively.

The ArEDC and the StEDC each control the execution of instructions in their respective execution units by referencing the Store Count Field and the instructions in the two instruction queues as shown by lines (57) and lines (58). Beginning with the instruction in the bottom position in the queue, each decision control checks whether one of the instructions in the bottom two positions of the queue can be executed. The decision control must delay the execution of an instruction only if it detects a dependency between the instruction and one of the instructions in the queue of the other execution unit.

The two decision controls use the store count associated with each instruction in the AQ to determine which instructions might be dependent. For example, the ArEDC knows that if the store count of an instruction is zero, no stores are in front of that instruction, and it can be executed. However, if the store count of an instruction is greater than zero, the ArEDC must check whether the instruction will write over the target of one of the store instructions that preceded it. The StEDC knows that if the store count of an arith-

metic instruction is not zero, the instruction was preceded by store instructions, and a store can be executed. On the other hand, if the store count of an arithmetic instruction is zero, the StEDC must check if the arithmetic will be writing to the target of the store instruction that it wants to execute. If not, the store instruction can be executed, but if the store is storing the target of the arithmetic instruction, the store must wait until that arithmetic instruction executes.

Fig. 3 is a flow chart of how the Store Execute Decision Control checks the instructions in the Store Instruction Queue. This figure shows the first few steps of checking the store instruction in SQ0. Initially, in step (100) the StEDC checks the store count of the arithmetic in AQ0. If the store count is not equal to zero, then the store came first in sequence, and the StEDC can go to step (102) and execute the store instruction. If the store count is equal to zero, then the arithmetic came first, and the StEDC must go to step (104) and compare the target register of the AQ0 arithmetic with the target register of the SQ0 store. If the targets match, then the store must be delayed in step (106). If the targets do not match, then the StEDC goes to step (108) and checks the store count of the instruction in AQ1. As before, if the store count is not equal to zero, then the store came first, and the StEDC can execute the store instruction in step (110). If the store count of the instruction in AQ1 is zero, then the store decision control must check whether or not the target register of the arithmetic in AQ1 is the same as the target register of the store in SQ0. If they are the same, the StEDC must go to step (114) and delay execution. If not, the StEDC continues to the arithmetic in AQ2. This process continues until all the instructions in the arithmetic queue have been checked. If the store target register does not match any of the AQ target registers, the store can be executed even though it followed arithmetics in the original sequence.

Fig. 4 is a flow chart showing a condensed version of Fig. 3 for checking all of the instructions in the Store Instruction Queue. In Fig. 4, the variable "x" stands for the position in the Arithmetic Instruction Queue and runs from 0 to 7. The variable "n" stands for the position in the Store Instruction Queue and also runs from 0 to 7. In the first iteration of step (120), "n" is zero for SQ0, and "x" is zero for AQ0. In step (120), the StEDC compares the store count of the arithmetic in AQx with the number, plus one, of the position of the store that it is checking for execution. For example, if the StEDC were checking the store in SQ1, then it would compare the store count of the instruction in the AQ with the number "2". If the store count is greater than or equal, then the store instruction came before the arithmetic instruction, and the store can be executed in step (122). If the store count is not greater than or equal, then the arithmetic instruction came first, and the StEDC goes to step

(124). In step (124), The StEDC compares the target registers of the two instructions. If the two instructions have the same target register, the StEDC must go to step (126) and delay the store. If the target registers are not the same, then the StEDC goes to step (128) and increments "x". The StEDC then continues to step (120) now using the next instruction in the arithmetic instruction queue. The StEDC goes through this process for the first two stores in the SQ and executes the first store that is executable.

Fig. 5 is a flow chart of how the Arithmetic Execute Decision Control checks the instructions in the Arithmetic Instruction Queue. This figure shows the first few steps of checking the arithmetic instruction in AQ0. In step (200), the ArEDC checks whether the store count of the instruction in AQ0 is zero. If the store count is zero, the ArEDC goes to step (202) and executes the arithmetic in AQ0 because the arithmetic has no stores that precede it. If the store count is not equal to zero, then the ArEDC goes to step (204). Because a store precedes the arithmetic instruction, the ArEDC must compare the targets of the arithmetic in AQ0 and the store in SQ0. If the targets compare, the arithmetic must be delayed in step (206). If the targets do not compare, then the ArEDC goes on to step (208) and checks whether the store count of the instruction in AQ0 is equal to one. If it is equal to one, the ArEDC can execute the arithmetic in step (210) because the decision control has already checked the one preceding store in SQ0. If the store count is not equal to one, then the ArEDC must go to step (212). In step (212), the ArEDC compares the target of the arithmetic in AQ0 with the target of the store in SQ1. If the targets compare, then the ArEDC must go to step (214) and delay the arithmetic. If the targets do not compare, the ArEDC continues the process. If the arithmetic target register does not match any of the SQ target registers, the arithmetic can be executed even though it followed the stores in the original sequence.

Fig. 6 is a flow chart showing a condensed version of Fig. 5 for checking all of the instructions in the Arithmetic Instruction Queue. As in Fig. 4, the variable "x" stands for the position in the Arithmetic Instruction Queue and runs from 0 to 7. The variable "n" stands for the position in the Store Instruction Queue and also runs from 0 to 7. In the first iteration of step (220), "n" is zero for SQ0, and "x" is zero for AQ0. In step (220), the ArEDC compares the store count associated with the arithmetic in AQx with "n+1". If the store count is not greater than or equal to "n+1", then the store followed the arithmetic, and the arithmetic can be executed in step (222). If the store count is greater than or equal to "n+1", the ArEDC must continue to step (224) because the store came before the arithmetic instruction. In step (224), the ArEDC compares the target registers of the arithmetic and the store. If the two targets are the same, then the ArEDC

must go to step (226) and delay the arithmetic until the store executes. If the targets do not match, the ArEDC goes to step (228) and increments "n". The ArEDC returns to step (220) and does the whole thing over again with the next store instruction. The ArEDC goes through this process for the first two arithmetics in the AQ and executes the first arithmetic that is executable.

Fig. 7 is a block diagram showing an example of delaying a store instruction in the Store Execution Unit. The instruction stream is two adds followed by a store. The first add has register 2 as its target, the second add has register 4 as its target and the store has register 2 as its target (or source). As shown in Fig. 7, the first and second adds occupy the bottom two positions in the Arithmetic Instruction Queue, positions (60) and (62). The store occupies the bottom position in the Store Instruction Queue (70). The store counts associated with the arithmetic instructions are zero and are shown in positions (80) and (82) in the Store Count Field. The zero store count of the adds means that no stores preceded them. If the instructions were to be executed sequentially, both adds would execute before the store. In order for the store to execute before the adds, it must not have the same target register as the target of either of the two add instructions. Because the target of the add instruction in the bottom of the AQ is register 2, and the target of the store is register 2, the store must wait for the bottom add to execute.

Fig. 8 shows an example of a store instruction in the Store Execution Unit being executed in front of an instruction in the Arithmetic Execution Unit that originally preceded it. The instruction stream shown is two adds followed by a store. The first add has as its target register 2, the second add has as its target register 4, and the store has as its target register 3. The two adds occupy the bottom two positions in the arithmetic queue and the store occupies the bottom position in the store queue. As shown, the store count of both of the add instructions is zero. This means that if the store is to execute before one or both of the two adds, it must not have the same target register as the target of one of the two add instructions. Because the store and the two adds do not have the same target registers, the store can execute before the two adds, which is out of order in the original sequence.

Fig. 9 shows an example of an add instruction being delayed in the Arithmetic Execution Unit because of a preceding store. It also shows an add instruction executed out of sequence with another add in the Arithmetic Execution Unit. The instruction sequence in Fig. 9 is a store followed by two adds. The store has as its target register 2, the first add has as its target register 2, and the second add has as its target register 4. The two add instructions occupy the bottom positions in the Arithmetic Instruction Queue and the store occupies the bottom position in the Store

Queue. The store count associated with each of the arithmetic instructions is one because there is one store preceding them. If either of the add instructions is to be executed, its target must be compared against the target of the store instruction. Because the add to target register 2 has the same target as the store, it must be delayed and wait for the store to execute. There is, however, no data dependency between the second add and the store, so the second add can execute. Thus, an add can be executed out of sequence with another add, as well as out of sequence with a store that preceded it.

Fig. 10 is a block diagram of an example of an arithmetic instruction in the Arithmetic Execution Unit being executed prior to a store instruction in the Store Execution Unit that came before it. The instruction stream is a store followed by two adds. The target registers are register 2, register 3, and register 4, respectively. The store count for each of the arithmetic instructions is one, because each has one store preceding it. The add in the bottom of the arithmetic queue can execute because its target register is 3, and the target register of the store in the store queue is 2.

Fig. 11 is a block diagram of an example of a store being executed out of sequence with another store that preceded it. The instruction stream is an add to register 2, an add to register 4, a store of register 2, and a store of register 3. The two stores occupy the bottom positions in the Store Instruction Queue and the two adds occupy the bottom positions in the Arithmetic Instruction Queue. The store count associated with each of the arithmetics is zero, because there are no stores that preceded the arithmetics in the original instruction stream. The store at the bottom of the store queue cannot be executed because the arithmetic at the bottom of the arithmetic queue also has a target register 2. Thus, the store must be delayed until the add executes. However, the store in the second position in the store queue can execute because there is no match. The store of register 3 can be executed before the store to register 2. Thus, a store can be executed out of sequence with another store, as well as out of sequence with an arithmetic that preceded it.

It should be apparent to one skilled in the art that this apparatus could be extended to allow an execution unit to execute instructions in its queue in any order as well as execute them out of sequence with instructions in another execution unit. The only delay occurs when an instruction to be executed in one unit must wait for an instruction in another unit to be executed.

While this invention has been described with reference to the illustrated embodiment, this description is not intended to be construed in a limiting sense. Various modifications of the illustrated embodiment, as well as other embodiments of the invention, will be

come apparent to those persons skilled in the art upon reference to this description. It is, therefore, contemplated that these appended claims will cover any modifications or embodiments as fall within the true scope of the invention.

Claims

1. A data processing system, comprising: a plurality of execution units, each including independent means for storing and for executing instructions; means for providing instructions from a sequence of instructions to said execution units, where each of said instructions is provided to only one of said execution units; means, connected to said execution units, for detecting when an instruction in a first execution unit must complete execution prior to execution of an instruction in a second execution unit to produce correct results; and means, responsive to said means for detecting, for delaying executing said instruction in said second execution unit until said instruction in said first execution unit has completed execution.
2. A data processing system according to Claim 1, wherein said instructions each belong to one of a plurality of classes of instructions including arithmetic, branch, load, store, and system control instructions, and each of said execution units is dedicated to executing one of said classes of instructions.
3. A data processing system according to Claim 2, further comprising means, connected to said means for providing, for determining the class of each instruction in said sequence of instructions.
4. A data processing system according to any preceding claim further comprising means for associating with each instruction in a first execution unit a count of instructions in a second execution unit that preceded said each instruction in said sequence.
5. A data processing system according to Claim 4, wherein said means for delaying includes: means for delaying executing an instruction in said first execution unit only if the count associated with said instruction is non-zero and an instruction in said second execution unit that preceded said instruction must execute prior to said instruction to produce correct results; and means for delaying executing an instruction in said second execution unit only if the count associated with an instruction in said first execution unit is less than or equal to a number of instructions in said second execution unit that precede said instruction in

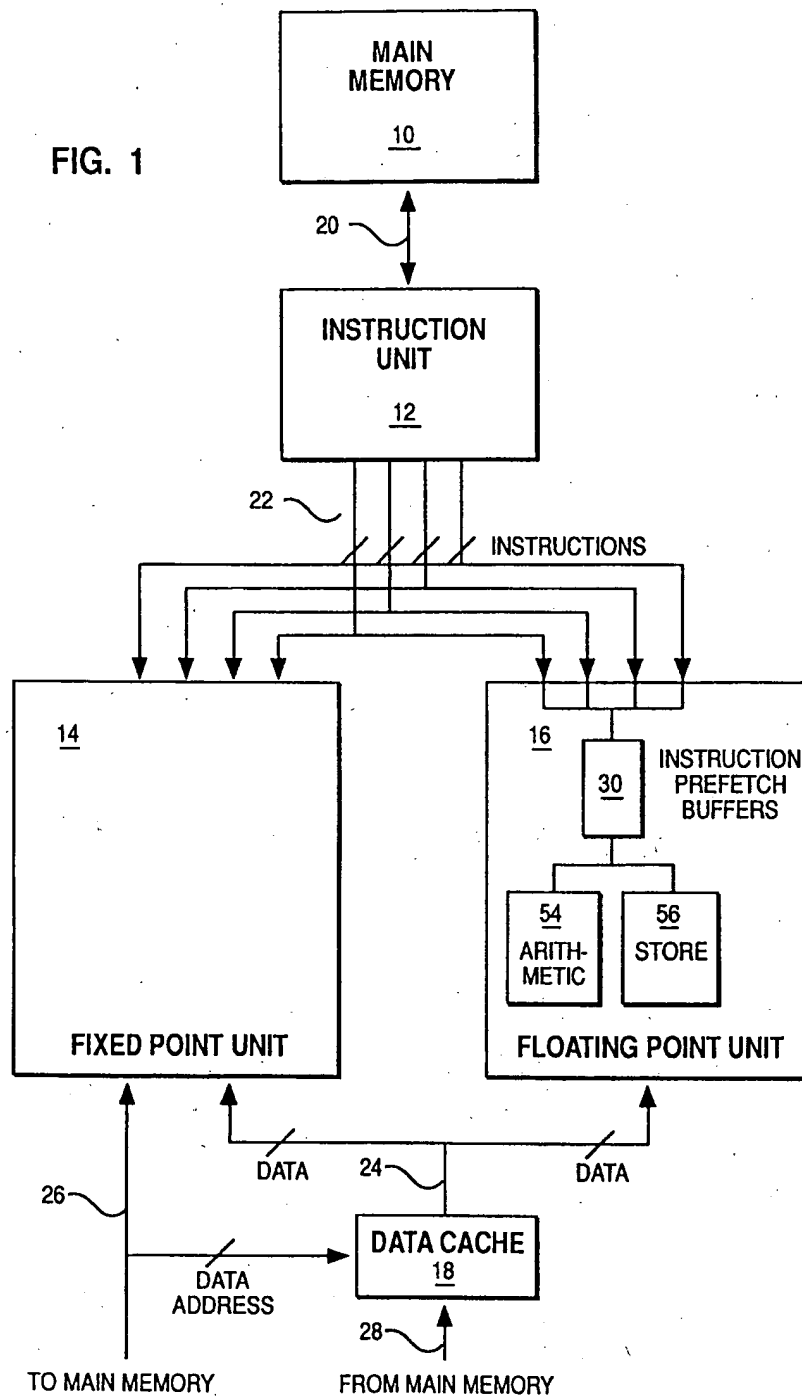
said second execution unit and said instruction in said first execution unit must execute prior to said instruction in said second execution unit to produce correct results.

6. A data processing system according to Claim 5, wherein a first class of instructions is floating point arithmetic instructions and said first execution unit is dedicated to executing said first class of instructions, and a second class of instructions is floating point store instructions and said second execution unit is dedicated to executing said second class of instructions.
7. In a data processing system, a method of processing comprising the steps of: providing instructions from a sequence of instructions to a plurality of execution units, where each of said instructions is provided to only one of said execution units; storing and executing instructions independently in said plurality of execution units; detecting when an instruction in a first execution unit must complete execution prior to execution of an instruction in a second execution unit to produce correct results; and delaying executing said instruction in said second execution unit until said instruction in said first execution unit has completed execution in response to said step of detecting.
8. A method of processing according to Claim 7, wherein said instructions each belong to one of a plurality of classes of instructions including arithmetic, branch, load, store, and system control instructions and each of said execution units is dedicated to executing one of said classes of instructions.
9. A method of processing according to any preceding claim, further comprising the step of associating with each instruction in a first execution unit a count of instructions in a second execution unit that preceded said each instruction in said sequence.
10. A method of processing according to Claim 9, wherein said step of delaying is carried out by: delaying executing an instruction in said first execution unit only if the count associated with said instruction is non-zero and an instruction in said second execution unit that preceded said instruction must execute prior to said instruction to produce correct results; and delaying executing an instruction in said second execution unit only if the count associated with an instruction in said first execution unit is less than or equal to a number of instructions in said second execution unit that precede said instruction in said second exe-

cution unit and said instruction in said first execution unit must execute prior to said instruction in said second execution unit to produce correct results.

11. A method of processing according to Claim 10, wherein a first class of instructions is floating point arithmetic instructions and said first execution unit is dedicated to executing said first class of instructions, and a second class of instructions is floating point store instructions and said second execution unit is dedicated to executing said second class of instructions.

FIG. 1



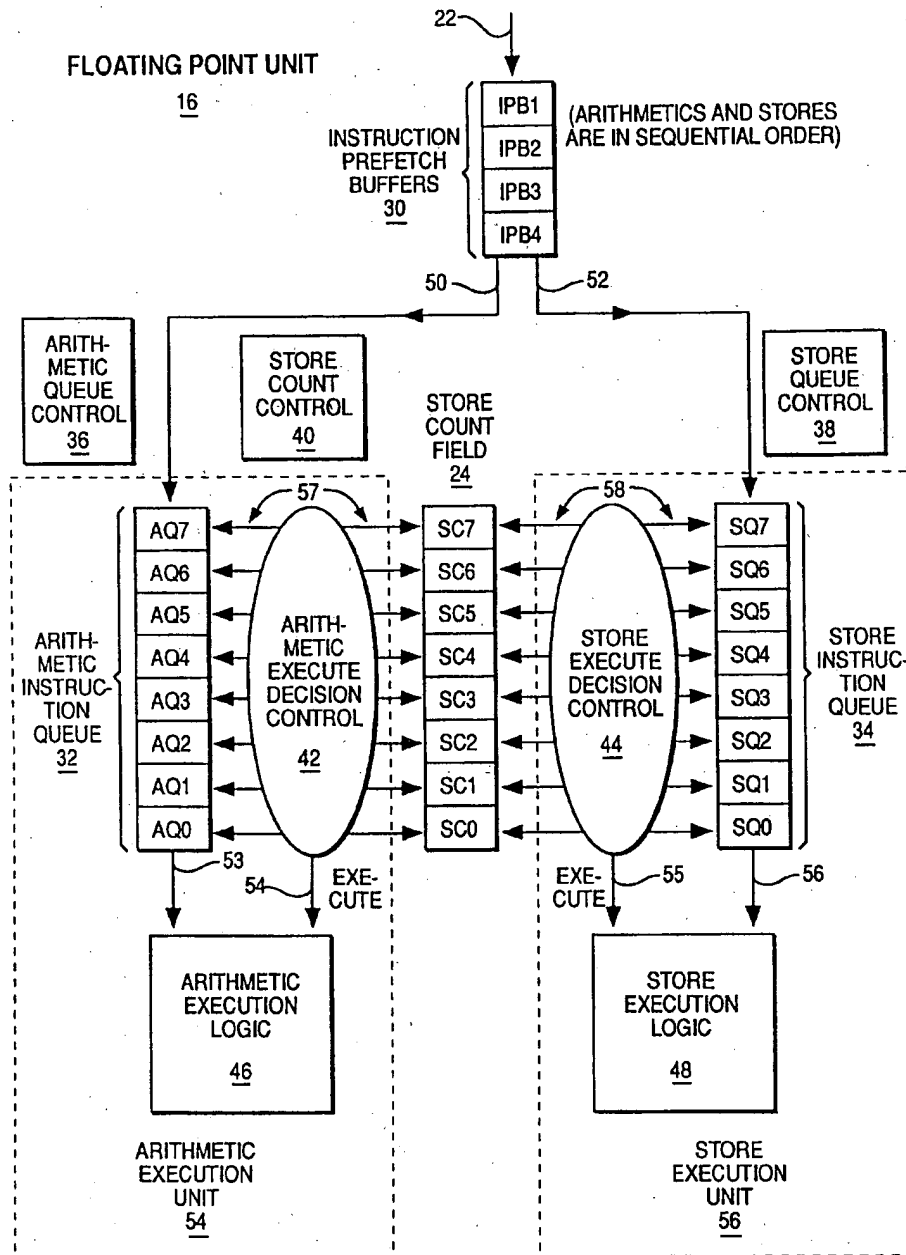
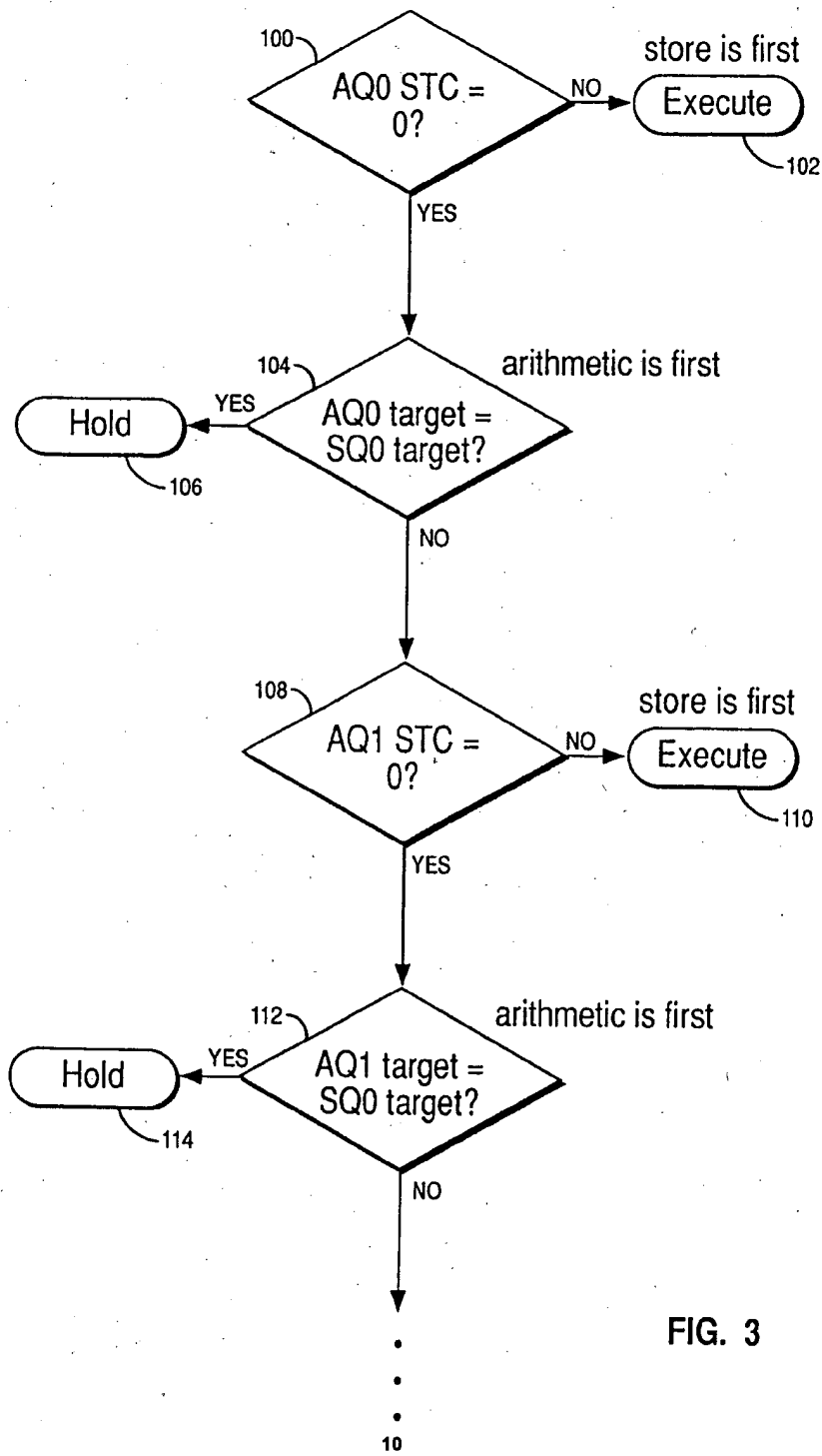
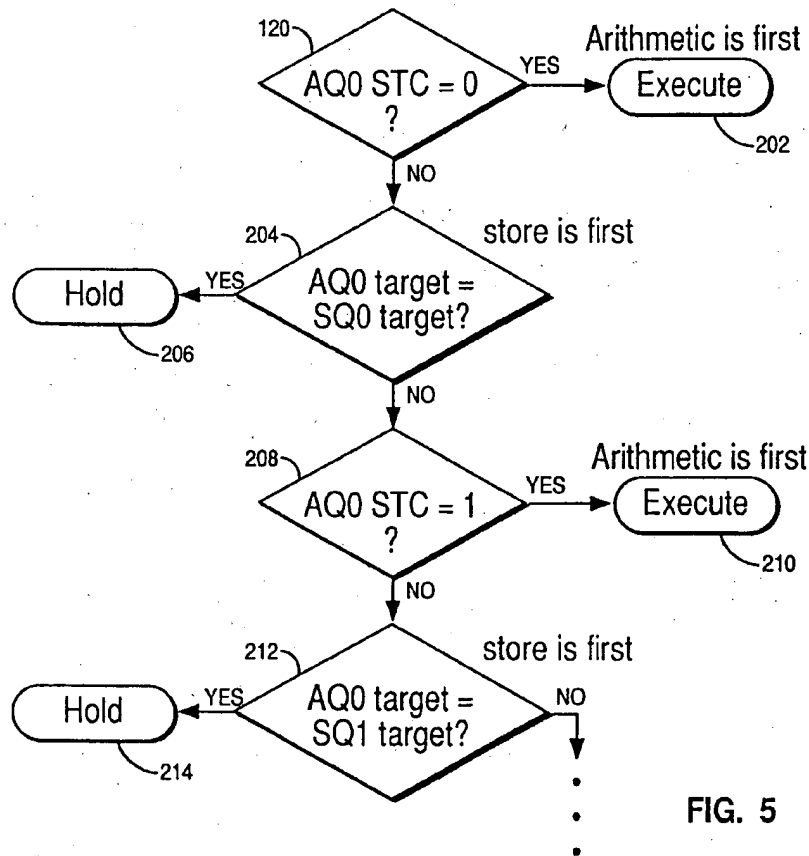
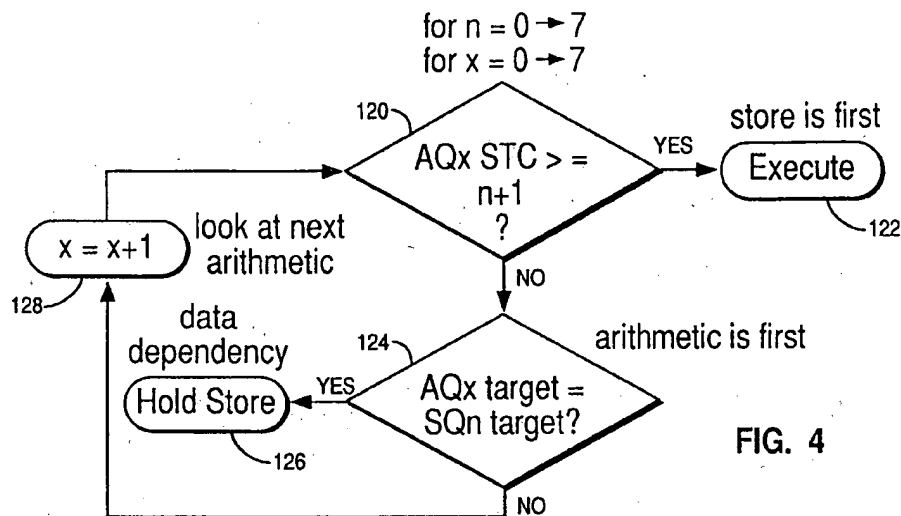
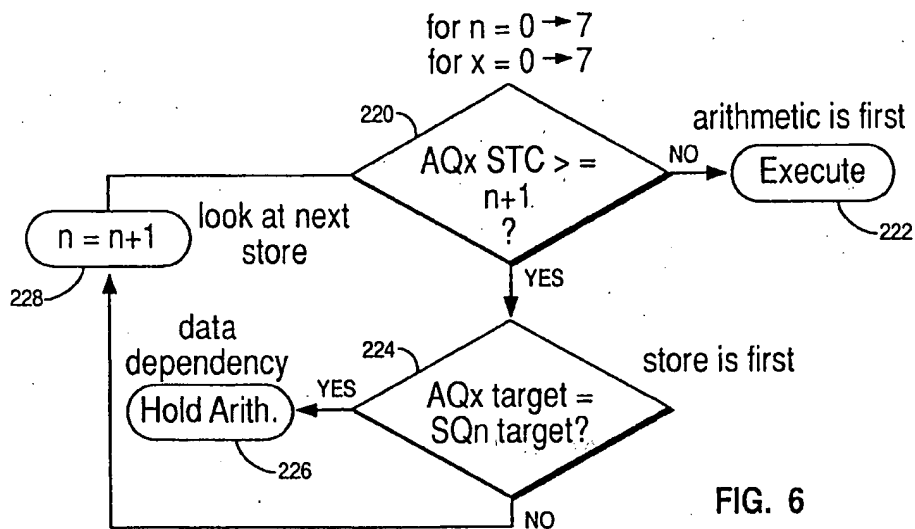


FIG. 2







INSTRUCTION SEQUENCE

add $2 \leftarrow 0 + 1$
add $4 \leftarrow 1 + 3$
store 2

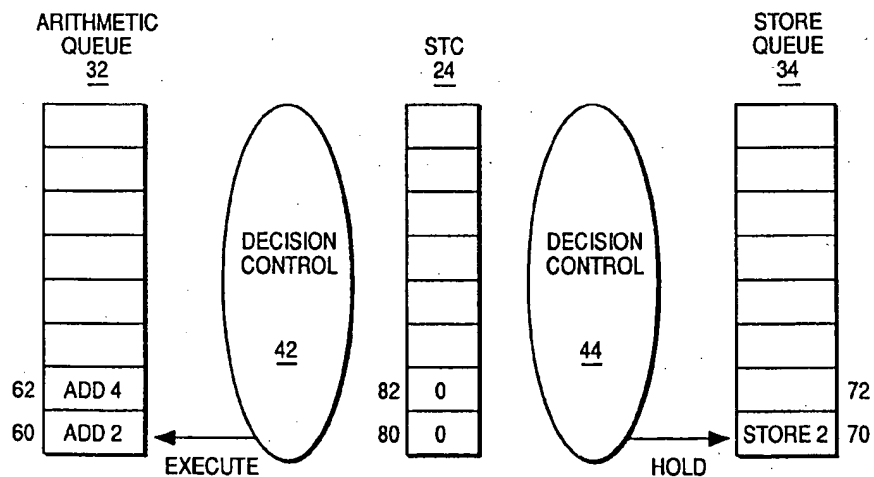


FIG. 8

INSTRUCTION SEQUENCE

add 2 \leftarrow 0 + 1
add 4 \leftarrow 1 + 3
store 3

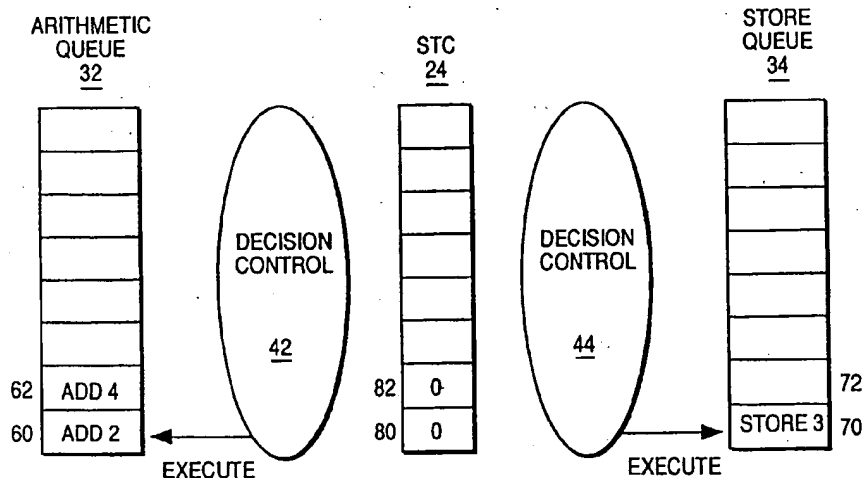


FIG. 9

INSTRUCTION SEQUENCE

store 2
add 2 \leftarrow 0 + 1
add 4 \leftarrow 1 + 3

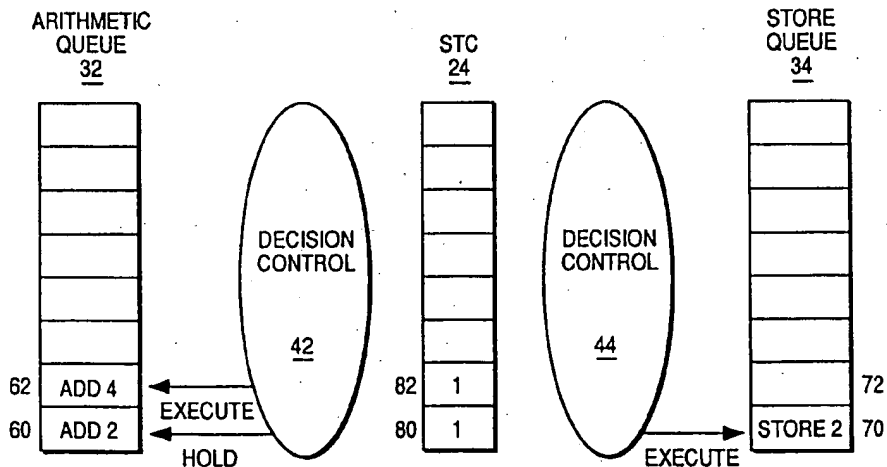


FIG. 10

INSTRUCTION SEQUENCE

store 2
add 3 $\leftarrow 0 + 1$
add 4 $\leftarrow 1 + 3$

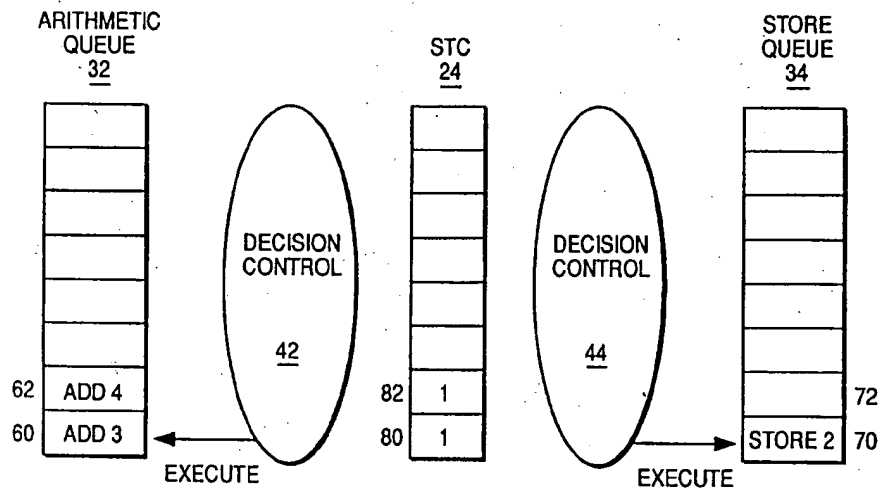


FIG. 11

INSTRUCTION SEQUENCE

add 2 $\leftarrow 0 + 1$
add 4 $\leftarrow 1 + 3$
store 2
store 3

